# **Optimizing Memory Access Patterns for High-Performance Computing: A Python Implementation and Report**

Jacob Jeffers

MSCS 532

Dr. Vanessa Cooper

Fall 2024

12/8/2024

**Optimizing Memory Access Patterns for High-Performance Computing: A Python Implementation and Report**

High-performance computing (HPC) is a compelling technology that solves complex problems at speeds far beyond those of conventional desktop, laptop, or server systems (Susnjara & Smalley, 2024). Operating at more than one million times faster than everyday computing systems, HPC becomes indispensable for tasks demanding immense computational power. For instance, supercomputers like the US-based Frontier, capable of processing 1.206 quintillion floating-point operations per second, are prime examples of what HPC can achieve (Susnjara & Smalley, 2024). HPC systems rely on various optimization techniques that maximize resource utilization, enhance memory access patterns, and enable efficient parallel task execution (Azad et al., 2021). These optimizations enable HPC to tackle workloads beyond human capability, such as sequencing DNA, automating stock trading, and processing AI data for self-driving vehicles.

This report will focus on memory access optimization, a crucial aspect of HPC performance. Efficient memory access is vital because it reduces latency and ensures that data is available when needed, thus preventing slowdowns in computation. Without optimization, memory bottlenecks may occur, even in the most powerful systems, significantly impeding performance. The research by Azad et al. (2021) outlines several key strategies to overcome these bottlenecks, including enhancing data locality, optimizing cache performance, and enabling parallel memory access. These methods boost the speed and efficiency of HPC systems and enable them to handle larger datasets and more intricate tasks, which drives advances in fields like genomics and AI.

Optimization is essential for HPC because it impacts memory access, data storage, and computational speed (Rompf et al., 2013). Poorly chosen data structures or neglecting optimization can lead to increased latency, inefficient cache usage, and unnecessary memory overhead—factors that detract from HPC performance. To address these concerns, techniques like multi-stage programming and algorithmic improvements, as noted by Stanford University, help minimize overhead and allow HPC systems to handle tasks like matrix manipulations and database queries more effectively (Rompf et al., 2013). Such improvements enable HPC to address large-scale challenges, from scientific research to AI-driven applications.

Memory access optimization is especially critical in HPC, as it targets the performance bottlenecks that can slow down workflows. In these systems, inefficient memory access increases latency, causes cache misses, and leads to data retrieval inefficiencies. These issues become particularly apparent in data structure-intensive tasks, such as matrix multiplication, where large amounts of data must be quickly accessed and processed. By optimizing memory access patterns, HPC systems can reduce cache misses and improve throughput, ensuring that data is retrieved most efficiently (Rompf et al., 2013; Azad et al., 2021). This efficiency is essential to maintaining the rapid performance HPC is known for, especially when working with vast datasets or performing complex calculations.

In scientific computing, for example, simulations that model complex physical systems demand substantial memory bandwidth and efficient handling of vast data sets within constrained time limits. Similarly, AI applications rely on optimized memory access to process massive datasets and execute complex algorithms effectively (Azad et al., 2021). Without effective memory access optimization, these systems would face slowdowns, preventing them from achieving the rapid processing speeds necessary for breakthroughs like climate modeling, genomics, and artificial intelligence.

**Code Implementation**

As discussed earlier, memory access optimization is essential for maintaining the high computational speeds of HPC. A practical way to demonstrate the impact of memory optimization is through tasks like matrix multiplication, which is a standard operation in fields such as scientific computing, AI, and machine learning (Fawzi et al., 2022). Large datasets need to be processed efficiently in these scenarios, and poor memory access can significantly slow down performance.

Matrix multiplication is a task that involves extensive data handling, where each element of the resulting matrix requires multiple reads and writes from memory (Dijkinga, 2024). Two implementations will be compared - an unoptimized code using Nested Loops and an optimized code using NumPy's dot() function. This example will demonstrate how optimized memory access can dramatically reduce execution time and make HPC systems more efficient.

***Unoptimized Implementation***

The unoptimized approach is a simple method that uses three nested loops to multiply matrices. While this method is straightforward, it becomes inefficient for larger matrices due to increased computation time and poor memory usage (*Python Program to Multiply Two Matrices*, n.d.). A snippet of the unoptimized implementation can be found below, with comments and a snippet of the Time it takes the code to multiply a 100X100 matrix.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

***Optimized Implementation***

Using NumPy's dot() function, the optimized approach leverages efficient memory handling and computational optimization due to NumPy being backed by optimized C libraries. This method ensures that data is processed in a more optimal order, reducing cache misses and improving throughput (GeeksforGeeks, 2022). By optimizing memory access and leveraging low-level optimizations, NumPy's approach significantly reduces the Time needed for matrix multiplication, particularly as the matrix size increases. A snippet of the unoptimized implementation can be found below, with comments and a snippet of the Time it takes both codes to multiply a 100X100 matrix.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

***Performance Analysis***

To compare the performance of these two methods, Python's time module will be used to measure the execution times of both implementations on matrices of varying sizes, such as 100x100, 500x500, and 1000x1000. The results will be displayed in a table, illustrating how the optimized method reduces execution time as the size of the matrices grows. A snippet of the code used to run the test and the table summarizing the results can be found below.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

|  |  |  |
| --- | --- | --- |
| Size | Unoptimized Time (s) | Optimized Time (s) |
| 100 | 0.04837 | 0.00032 |
| 200 | 0.39478 | 0.00080 |
| 500 | 10.35617 | 0.00290 |
| 1000 | 94.24775 | 0.01277 |

**Conclusion**

The results of the matrix multiplication tests demonstrate the benefits of memory access optimization in high-performance computing. The unoptimized implementation, which relied on basic nested loops, struggled significantly as the matrix size increased. While it performed reasonably well for smaller matrices, taking approximately 0.048 seconds for a 100X100 matrix, its performance deteriorated sharply for larger-sized matrices. For example, the largest matrix tested required over 94 seconds to complete the task, illustrating its inefficiency in handling large datasets.

In contrast, the optimized implementation using NumPy's dot() function showed remarkable efficiency. The optimized version was completed in under 0.013 seconds, even at the largest matrix size tested. This drastic reduction in execution time highlights the effectiveness of memory optimization and the computational benefits of leveraging optimized libraries like NumPy.

These findings underscore the importance of memory optimization in HPC. Optimized approaches allow systems to scale effectively and easily handle computationally intensive tasks.

GitHub: <https://github.com/jakejeffers/MSCS-532-Final-Project>

**References**

Azad, M. A. K., Iqbal, N., Hassan, F., Roy, P., & Department of Computer and Information Science, University of Michigan - Dearborn. (2021). An empirical study of High-Performance computing (HPC) performance bugs. In *University of Michigan - Dearborn*. <https://foyzulhassan.github.io/files/MSR23_HPC.pdf>

Dijkinga, F. J. (2024, November 16). Large Matrix Multiplication with Python: A Data Scientist's Guide. *Medium*. <https://medium.com/@fernando.dijkinga/large-matrix-multiplication-with-python-a-data-scientists-guide-25b2622d3c1c>

Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., Ruiz, F. J. R., Schrittwieser, J., Swirszcz, G., Silver, D., Hassabis, D., & Kohli, P. (2022). Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, *610*(7930), 47–53. <https://doi.org/10.1038/s41586-022-05172-4>

GeeksforGeeks. (2022, November 18). *Numpy. Dot () in Python*. GeeksforGeeks. <https://www.geeksforgeeks.org/numpy-dot-python/>

*Python program to multiply two matrices*. (n.d.). <https://www.programiz.com/python-programming/examples/multiply-matrix>

Rompf, T., Sujeeth, A. K., Amin, N., Brown, K. J., Jovanovic, V., Lee, H., Jonnalagedda, M., Olukotun, K., & Odersky, M. (2013). Optimizing data structures in high-level programs. *ACM SIGPLAN Notices*, *48*(1), 497–510. <https://doi.org/10.1145/2480359.2429128>

Susnjara, S., & Smalley, I. (2024, August 22). HPC. *What is high-performance computing (HPC)?* Retrieved December 5, 2024, from <https://www.ibm.com/topics/hpc>